CLEAR: Context Augmentation from Contrastive Learning of Experience via Agentic Reflection

Linbo Liu **Guande Wu** **Han Ding** **Yawei Wang** **Qiang Zhou**
Yuzhe Lu **Zhichao Xu** **Huan Song** **Panpan Xu** **Lin Lee Cheong**
AWS AI
Correspondence: linbol@amazon.com

arXiv:2604.07487v3 [cs.AI] 21 Oct 2026

Abstract

Large language model agents rely on effective model context to obtain task-relevant information for decision-making. Many existing context engineering approaches primarily rely on the context generated from the past experience and retrieval mechanisms that reuse these context. However, retrieved context from past tasks must be adapted by the execution agent to fit new situations, placing additional reasoning burden on the underlying LLM. To address this limitation, we propose a generative context augmentation framework using **C**ontrastive **L**earning of **E**xperience via **A**gentic **R**eflection (**CLEAR**). **CLEAR** first employs a reflection agent to perform contrastive analysis over past execution trajectories and summarize useful context for each observed task. These summaries are then used as supervised fine-tuning data to train a context augmentation model (CAM). Then we further optimize CAM using reinforcement learning, where the reward signal is obtained by running the task execution agent. By learning to generate task-specific knowledge rather than retrieve knowledge from the past, CAM produces context that is better tailored to the current task. We conduct comprehensive evaluations on the AppWorld and WebShop benchmarks. Experimental results show that **CLEAR** consistently outperforms strong baselines. It improves task completion rate from 72.62% to 81.15% on AppWorld test set and averaged reward from 0.68 to 0.74 on a subset of WebShop, compared with baseline agent. Our code is publicly available at https://github.com/awslabs/CLEAR.

1 Introduction

Large language models (LLMs) have become increasingly powerful as their parameter scale continues to grow (Xi et al., 2025; Kaplan et al., 2020; Wei et al., 2022a; Ding et al., 2024). Recent works have demonstrated that LLMs can act as agents in sequential decision-making settings and achieve strong performance across a variety of tasks (Yao et al., 2023b; Talebirad and Nadiri, 2023; Wang et al., 2024b; Xia et al., 2025; Black et al., 2024; Hu et al., 2025). Despite these advances, LLMs still rely primarily on parametric knowledge stored in their model weights when performing reasoning, which can be outdated, incomplete, or insufficient for complex, knowledge-intensive tasks (Lewis et al., 2020; Gao et al., 2023; Suzgun et al., 2025). Effective integration of external knowledge and task-relevant context remains a key challenge in improving agent decision-making capabilities.

Retrieval augmented generation (RAG) (Lewis et al., 2020) and other context engineering techniques (Mei et al., 2025) are proposed to bridge the gap between parametric knowledge and context integration. However, typical RAG systems face several practical challenges, including designing effective knowledge base indexing strategies (Huang et al., 2025), query rewriting (Ma et al., 2023; Peng et al., 2024; Xu et al., 2025b), and devising reliable retrieval pipelines (Jin et al., 2025; Shao et al., 2023; Yu et al., 2022; Xu et al., 2026). Moreover, their performance heavily depends on the quality and relevance of the underlying knowledge base. Several recent works on prompt optimization attempt to address these limitations. For example, Agentic Context Engineering (ACE) (Zhang et al., 2025) and Dynamic Cheatsheet (Suzgun et al., 2025) learn instructions from the past experience of LLM agents and reuse them to assist decision-making on future tasks. GEPA (Agrawal et al., 2025) proposes an iterative prompt optimization framework using pareto-based candidate selection. (Li et al., 2024) trains a prompt rewriter to generate the best prompt. However, these universally learned guidance or optimized prompts are typically static and general-purpose, rather than tailored to specific future task instance. Consequently, the execution agent must reason about how to adapt them to the current task. This requirement can be-

come problematic when the underlying LLM has limited reasoning capability, or when the future task differs substantially from previous ones, in which case the stored guidance and optimized prompts may be only weakly relevant. See Section C for a detailed discussion.

To address this limitation, we propose a context augmentation framework using **C**ontrastive **L**earning of **E**xperience via **A**gentic **R**eflection (**CLEAR**). **CLEAR** first employs a reflection agent to perform contrastive analysis over past experience replays and summarize useful context for each task. The resulting context is then used as supervised fine-tuning (SFT) data to train a context augmentation model (CAM). After the SFT stage, we further optimize the CAM using reinforcement learning (RL), where the reward signal is obtained by executing the task execution agent (see Figure 1). Further, the choice of the CAM can be lightweight and agnostic to the choice of the expensive execution agent, adding negligible overhead to the overall system.

The trained CAM provides additional context that is useful for solving future tasks, which will be integrated into the prompt of the task execution agent, as shown in Figure 2. Importantly, **CLEAR** does not require parametric training of the underlying LLM for execution agents, which are often proprietary models with no access to their weights. Instead, **CLEAR** only requires training the smaller CAM. As a result, **CLEAR** is a unified framework that can be applied to LLM agents built on either proprietary or open-source foundation models.

Our **CLEAR** framework has the following contributions:

- We propose a CAM that generates additional context to improve the performance of LLM agents. CAM integrates task-relevant context into the prompt of the execution agent, avoiding any modification of the underlying LLM weights and making the framework broadly applicable across different agentic systems.
- For CAM training data generation, we introduce an agentic reflection mechanism that performs contrastive learning over past execution trajectories. By systematically analyzing multiple trajectories, the reflection agent extracts high-quality instructions as SFT training data for CAM.
- We design a two-stage training pipeline (SFT + RL) to train the CAM. In particular, we build a novel RL training framework that couples the CAM with the execution agent to generate rollouts, while only updating CAM's parameters.
- We evaluate **CLEAR** against diverse context engineering methods, including RAG and ACE, across multiple benchmarks. **CLEAR** consistently outperforms all baselines on every evaluated dataset.

2 Related Work

LLM Agents. LLM agents extend foundation models into autonomous, goal-directed systems by augmenting them with planning, memory, tool use, and action modules (Wang et al., 2024a; Xi et al., 2025). LLM agents are built upon the foundational reasoning capabilities, demonstrated by works such as CoT (Wei et al., 2022b), ReAct (Yao et al., 2023b), ToT (Yao et al., 2023a), and DoT (Lingam et al., 2025). When equipped with tools, LLMs can learn to invoke external APIs to overcome inherent limitations in calculation, retrieval, and real-world interaction (Schick et al., 2023; Patil et al., 2024; Qian et al., 2025). Later, Anthropic's Model Context Protocol (MCP) (Anthropic, 2025) proposed a standardized open protocol for connecting LLM agents to external tools and data sources, addressing the fragmentation of tool integration interfaces. Browser-use and terminal-use agents have also matured, with Operator (OpenAI, 2025) and Claude Code[1] demonstrating agents that autonomously navigate web browsers and terminal environments to complete real-world tasks.

On the multi-agent front, several frameworks have demonstrated that collaboration among specialized LLM agents can tackle complex tasks more effectively than single agents. CAMEL (Li et al., 2023) explored role-playing-based cooperative communication, while MetaGPT (Hong et al., 2024) and ChatDev (Qian et al., 2024) organized agents into software-engineering teams following structured workflows. AutoGen (Wu et al., 2024) provided a general-purpose multi-agent conversation framework with human-in-the-loop support.

Contrastive Learning. Contrastive signals are widely used to learn robust representations by explicitly comparing informative alternatives (Gutmann and Hyvärinen, 2010; Ma and Collins, 2018;

[1] https://github.com/anthropics/claude-code

van den Oord et al., 2019; Zhang and Stratos, 2021; Xu et al., 2025a). In LLM-agent settings, a closely related idea appears in reflection-based methods that learn from behavioral differences across trials, especially between successful and failed executions (Shinn et al., 2023; Wang et al., 2023; Yu et al., 2026; Forouzandeh et al., 2025; Allard et al., 2026). Our work applies this principle in a practical agent-training pipeline: we contrast multiple rollouts for the same task and distill reusable strategy-level context through an agentic reflector. The novelty is mainly in this task-level adaptation and integration with context augmentation, rather than in a new contrastive objective itself.

Context Engineering. Context engineering aims to provide LLM agents with task-relevant information at inference time. Retrieval-based methods such as RAG (Lewis et al., 2020; Gao et al., 2023) improve factual grounding by retrieving external evidence, and generate-then-read variants further combine parametric generation with retrieval to improve coverage (Yu et al., 2022). Recent agent-centric approaches, including ACE (Zhang et al., 2025) and MIPRO (Opsahl-Ong et al., 2024), maintain evolving playbooks distilled from prior executions, while GEPA (Agrawal et al., 2025) optimizes prompts through reflective evolution. Compared with these methods, **CLEAR** learns a dedicated context augmentation model that maps a new task directly to actionable context via SFT+RL, rather than relying on nearest-neighbor retrieval or purely prompt-level updates. This shifts more adaptation into a trainable model and reduces the burden on the execution agent to reinterpret retrieved past experience.

LLM Fine-Tuning. The dominant paradigm for post-training large language models follows a two-stage pipeline: SFT on curated demonstrations, followed by RL to further align model behavior with desired objectives (Ouyang et al., 2022; Bai et al., 2022). The RL stage has been realized through various algorithms, including PPO (Schulman et al., 2017), which optimizes a clipped surrogate objective with a learned value function; DPO (Rafailov et al., 2023), which bypasses explicit reward modeling by directly optimizing on preference pairs; GRPO (Shao et al., 2024), which computes group-relative advantages to eliminate the critic model entirely; and many others (Zhang et al., 2021; Ahmadian et al., 2024; Yu et al., 2025; Yue et al., 2025; Zheng et al., 2025). **CLEAR** follows the same two-stage training paradigm of SFT followed by RL and is agnostic to the choice of RL algorithm. In our experiments, we adopt GRPO for policy optimization.

3 Preliminaries

3.1 LLM Reinforcement Learning.

The application of reinforcement learning to LLMs became popular with reinforcement learning from human feedback (RLHF) (Ouyang et al., 2022; Leike et al., 2018; Askell et al., 2021; Bai et al., 2022; Rafailov et al., 2023). In this framework, an LLM is modeled as a stochastic policy $\pi_\theta(y \mid x)$ that generates tokens autoregressively. Human preference data are first collected to train a reward model, which is then used to optimize the policy via reinforcement learning.

More recently, RL-based post-training has been extended to enhance reasoning ability, tool use, and long-horizon decision-making in agentic settings. These approaches frame text generation as sequential decision-making with downstream task rewards, rather than purely token-level prediction, as introduced in Section 3.2.

3.2 LLM Agent for Decision Making.

As LLMs grow increasingly capable, deploying them as agents in sequential decision-making settings has become a prominent research direction (Nakano et al., 2021; Yao et al., 2023b). We formalize this setting by modeling an LLM agent as a policy π within a Partially Observable Markov Decision Process (POMDP), defined by the tuple

$$M = (\mathcal{S}, \mathcal{A}, \mathcal{O}, P, R, \gamma), \tag{1}$$

where $\mathcal{S}$ is the latent state space, $\mathcal{A}$ is the action space, $\mathcal{O}$ is the observation space, $P(s_{t+1} \mid s_t, a_t)$ is the transition dynamics, R is the reward function, and γ is the discount factor.

An initial task description q is sampled from the task distribution $q \sim \mathcal{D}$, which produces a state s_0. Based on q and historical observation, the agent makes an action and receives an observation from the environment. Denote the agent's history as

$$h_0 = q, \quad h_t = (q, a_0, o_1, a_1, \ldots, o_{t-1}, a_{t-1}, o_t).$$

At each time step t, the task execution LLM agent $\pi_\theta^E(a_t \mid h_t)$ consumes the full history h_t and produces the next action $a_t \in \mathcal{A}$. The environment then transitions to a new state s_{t+1} according to P and returns the subsequent observation $o_{t+1} \in \mathcal{O}$.

This process continues until the interaction terminates after T steps, yielding a complete episode trajectory

$$\tau = (q, a_0, o_1, a_1, \ldots, o_{T-1}, a_{T-1}, o_T). \quad (2)$$

Then a scalar reward $r = R(\tau)$ is assigned to the entire trajectory, reflecting the overall quality of the agent's behavior over the episode. In this work, we propose a context augmentation method to add auxiliary context c into q, so that the expected reward can be improved on $\mathcal{D}$.

4 Our Proposed Method

Many practical LLM-based agents (e.g., (Yang et al., 2024; Xia et al., 2025; Liu et al., 2025b)) are built on top of proprietary foundation models such as OpenAI GPT models (Singh et al., 2025), Anthropic Claude (Anthropic, 2025a,b), and Google Gemini (Team et al., 2023). Although these agentic systems are often deployed through open-source agent frameworks such as Strands Agents[2], LangGraph[3], and OpenHands (Wang et al., 2025), the underlying foundation models remain closed-source. As a result, their internal parameters are inaccessible, limiting the feasibility of weight-level adaptation.

In this work, we propose a unified context augmentation framework that operates without modifying the LLM agents' weights. Our approach is compatible with both proprietary models and open-source models such as Qwen (Yang et al., 2025), DeepSeek (Liu et al., 2025a), Olmo (Olmo et al., 2025), and Kimi (Team et al., 2025). Instead of updating model parameters, we improve agent performance by augmenting the context via contrastive learning from past experience. When a task execution agent $\pi^E(\cdot)$ performs a task q, we augment its task description q with additional context c produced by CAM. Formally, we define a *replay buffer*

$$\Gamma = \big\{(\tau_1, R(\tau_1)), \ldots, (\tau_n, R(\tau_n))\big\}$$

as a collection of past trajectories and their corresponding outcome rewards, where τ_i is defined in Equation (2). Let $q \sim \mathcal{D}$ be an initial task description sampled from task distribution $\mathcal{D}$. We define a *context augmentation model* that maps q to an auxiliary context c that is appended to q before action generation from π^E. In other words, execution agent π^E will have $q \oplus c$ as its task description, where $\oplus$ denote concatenation.

We define the CAM as $\pi^C_\theta(\cdot)$ parameterized by θ. Given a task q, the model generates additional context $c = \pi^C_\theta(q)$. Our objective is to learn an optimal $\pi^C_\theta(\cdot)$ from Γ such that the expected return of the task execution agent π^E is maximized on $\mathcal{D}_{\text{train}}$.

To achieve this objective, we propose **CLEAR** (**C**ontrastive **L**earning of **E**xperience via **A**gentic **R**eflection), a three-phase training framework that combines contrastive learning, agentic reflection, SFT, and RL to optimize the CAM π^C_θ. In Phase 0, we employ a reflection agent to perform contrastive analysis over past execution trajectories and generate training data for SFT. In Phase 1, we fine-tune an open-source LLM using SFT as a warm-up stage. In Phase 2, we further optimize the model via RL to directly maximize the expected return of the task execution agent:

$$\begin{aligned} &\max_\theta J(\theta) \\ &= \max_\theta \mathbb{E}_{q\sim\mathcal{D}_{\text{train}},\, c\sim\pi^C_\theta(q),\, \tau\sim\pi^E(\cdot|q\oplus c)}\big[R(\tau)\big]. \end{aligned} \quad (3)$$

Intuitively, this objective encourages the CAM to generate useful context that improves the execution agent's expected performance. A concurrent work (Asawa et al., 2026) designs a similar RL pipeline to train an advisor model, but doesn't perform contrastive learning using agentic reflection and SFT. However, as discussed in Section A, all three phases in **CLEAR** can bring non-trivial performance improvement, making it a more comprehensive framework for agent refinement. We now introduce **CLEAR** in details.

4.1 Agentic Reflection via Contrastive Learning

Learning from a single trajectory is insufficient for robust agent refinement. A single execution provides only a narrow and potentially noisy view of the decision process. Therefore, refinement should leverage *multiple trajectories* for the same task. See Section A for an ablation study.

To achieve this, we introduce a reflection agent π^R that performs contrastive analysis over the replay buffer Γ for data generation. Its objective is to extract high-value insights that explain the behavioral distinctions among multiple trajectories. To enable scalable analysis, the reflection agent

[2] https://strandsagents.com/latest/
[3] https://www.langchain.com/langgraph

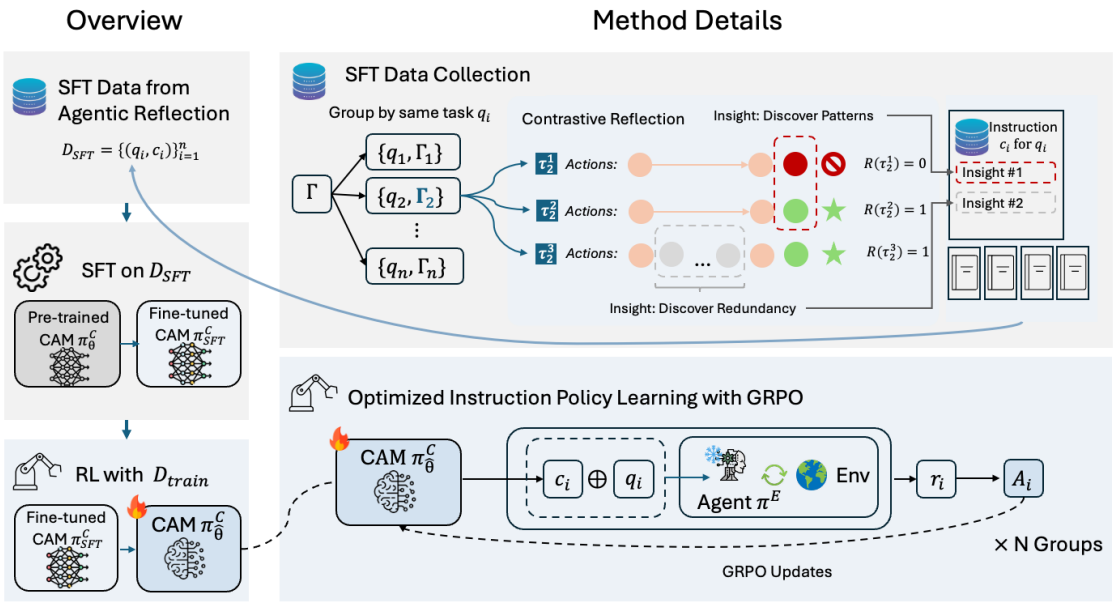


Figure 1: **CLEAR** training framework design. First, we execute each task $q_i \sim \mathcal{D}_{\text{train}}$ for m times and collect groups of replay Γ_i for q_i. We employ reflection agents π^R to generate instance-level instruction c_i for each q_i, collected into $\mathcal{D}_{\text{SFT}}$. Then, we initialize CAM from an open-source LLM and perform SFT on $\mathcal{D}_{\text{SFT}}$. Finally, we further perform RL on the trained CAM, which leverages the reward signal from π^E for policy update of CAM.

is equipped with a shell tool that allows it to selectively read trajectory files. This design is particularly important when trajectories are large and cannot be loaded entirely into context. We provide the prompts for π^R in Section F.

To fully leverage the benefits of contrastive analysis, we execute each task multiple times to obtain a set of trajectories corresponding to the same task instance. These trajectories capture diverse execution behaviors and outcomes, providing a rich resource for identifying task-specific decision patterns through contrastive comparison.

Specifically, for each task instance $q_i \sim \mathcal{D}_{\text{train}}$, we execute the task m times to obtain trajectories $\tau_i^1, \ldots, \tau_i^m$ and their corresponding rewards $r_i^1, \ldots, r_i^m$. These trajectories are then organized into a grouped replay buffer $\Gamma_i = \{(\tau_i^1, r_i^1), \ldots, (\tau_i^m, r_i^m)\}$. We obtain $c_i = \pi^R(\Gamma_i, q_i)$ by applying a reflection agent π^R to analyze the replay buffer Γ_i and summarize helpful context. Intuitively speaking, c_i can be viewed as an additional instruction to complete q_i. Up to this point, the generated pairs (q_i, c_i) form a high-quality SFT dataset, which will be used to train the augmentation model π_θ^C.

4.2 Training Framework

In this subsection, we adopt the standard post-training paradigm widely used in LLM alignment (Ouyang et al., 2022; Guo et al., 2025; Liu et al., 2025a; Yang et al., 2025): a two-stage framework consisting of SFT followed by RL.

SFT. Using the data collection pipeline described previously, we obtain a supervised dataset $\mathcal{D}_{\text{SFT}} = \{(q_i, c_i)\}_i$. We initialize the context augmentation model π_θ^C with a pre-trained LLM parameterized by θ. We then fine-tune the model on the supervised dataset $\mathcal{D}_{\text{SFT}}$ to obtain updated model π_{SFT}^C. The resulting model will be served as the initialization for the subsequent RL stage.

RL. In this phase, we further optimize π_{SFT}^C using reinforcement learning to directly maximize expected task reward. The training objective is introduced in Equation (3). Note that (i) In Equation (3), parameters θ in π_θ^C are the only trainable parameters, and the execution agent π^E will be frozen. (ii) The reward signal for π_θ^C is the same as the reward for running the execution agent π^E with $q \oplus c$, where $c \sim \pi_\theta^C(q)$. We optimize π_θ^C using policy gradient methods. Specifically in our experiments, we adopt GRPO as the policy optimization algorithm. See Figure 1 for an illustration of the

workflow.

4.3 Comparison to Existing Work

MIPRO (Opsahl-Ong et al., 2024), GEPA (Agrawal et al., 2025) and ACE (Zhang et al., 2025) are representative approaches for automatic prompt optimization. We refer to these methods as *the compared methods* throughout this section. While these approaches have demonstrated promising results, our **CLEAR** framework differs from *the compared methods* in several key aspects.

First, all *the compared methods* are training-free frameworks that relies entirely on off-the-shelf LLMs acting as the reflector and curator to generate the playbook and instructions. In contrast, **CLEAR** performs parametric learning: we train a context augmentation model π_θ^C using SFT followed by RL.

Second, the reflection agent π^R used in Phase 0 of **CLEAR** is inspired by *the compared methods* but differs substantially in design. Reflectors in *the compared methods* are implemented as a single LLM call, whereas our π^R is an agentic system equipped with tools for trajectory inspection and analysis. Moreover, our π^R explicitly performs contrastive reasoning over multiple trajectories to extract useful instructions, which is not a focus of *the compared methods*.

Third, the prompt templates used by *the compared methods* for the reflector and curator are benchmark-specific. For example, ACE's prompts include explicit instructions tailored to the AppWorld tasks and are not designed to generalize across benchmarks[4]. In contrast, the prompt template used by our reflection agent is general and benchmark-agnostic (see Section F). Despite ACE employing benchmark-specific prompt engineering for AppWorld, **CLEAR** consistently outperforms ACE as shown in Table 1.

We also discuss the comparison to RAG in Section C.

5 Experiments

To evaluate our **CLEAR** framework, we conduct experiments on the AppWorld (Trivedi et al., 2024) and WebShop (Yao et al., 2022) dataset.

5.1 Experiment Setting

We introduce our experiment setting for agentic data collection phase as follows and leave the training details of SFT and RL to Section E.

Execution Agent. We adopt the Strands Agents framework as the backbone of our agentic system. Strands Agents is a lightweight yet powerful SDK for building and deploying AI agents using a model-driven design paradigm. It supports a broad range of applications, from simple conversational assistants to complex autonomous workflows, and scales seamlessly from local development to production environments. We use `Claude-Sonnet-4` (Anthropic, 2025a) and `DeepSeek-V3.1` (Liu et al., 2024), accessed via Amazon Bedrock[5], as the foundation models for the execution agent π^E. We leverage the above agentic framework as the execution agent to run the training dataset of AppWorld and Webshop. To accelerate agent execution, we deploy the agent to Amazon Bedrock AgentCore[6] Runtime, which bootstraps multiple containers in parallel to support high-concurrency rollout execution. We set $m = 6$, i.e. for each task in the training set, we run the agent 6 times to collect trajectories. We then use their official evaluation harness to compute the outcome reward for each trajectory.

Reflection Agent. We use the Strands Agents framework together with `Claude-Sonnet-4` to build a reflection agent for contrastive analysis. The full prompt used for the reflection agent is provided in Section F. Furthermore, we leverage a combinatorial data augmentation technique to enlarge SFT dataset size if insufficient, as detailed in Section E.

CAM. The CAM $\pi_\theta^C(\cdot)$ is initialized from a `Qwen/Qwen3-32B` model (Yang et al., 2025), downloaded from HuggingFace. SFT and RL details can be found in Section E.

5.2 Baseline and Compared Methods

Baseline. We compare **CLEAR** with the untuned baseline using the execution agent $\pi_\theta^E(\cdot)$ and the initial task description q without any context augmentation.

RAG. To provide a stronger comparison, we also construct a RAG method. Specifically, we store all (q_i, c_i) pairs from $\mathcal{D}_{\text{SFT}}$ in a vector database, whose embedding is generated by `BAAI/bge-base-en-v1.5` (Xiao et al., 2023) from

[4] https://github.com/ace-agent/ace-appworld/tree/main/experiments/prompts

[5] https://aws.amazon.com/bedrock/

[6] https://aws.amazon.com/bedrock/agentcore/

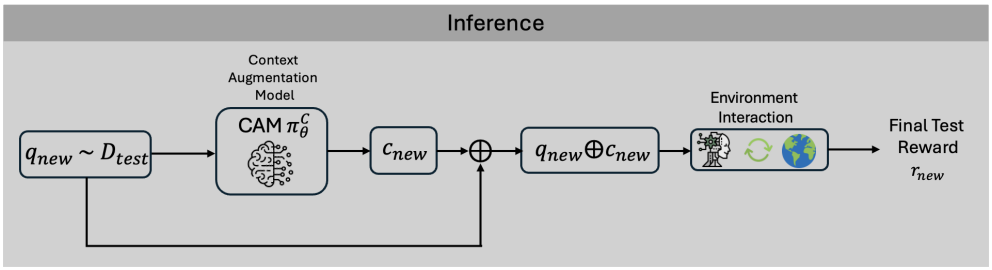


Figure 2: During inference, a new task q_{new} is sampled from $\mathcal{D}_{\text{test}}$ and is passed into π_θ^C to generate $c_{\text{new}} \sim \pi_\theta^C(q_{\text{new}})$. The auxiliary context c_{new} is appended to q_{new} and the execution agent π^E starts with $q_{\text{new}} \oplus c_{\text{new}}$.

Method	TGC ↑		SGC ↑	
	Avg	Pass@3	Avg	Pass@3
Baseline	72.62(2.59)	86.90	52.38(2.73)	66.07
RAG	72.02(2.15)	86.31	54.67(3.72)	71.43
ACE (Zhang et al., 2025)	74.40(3.57)	85.71	58.93(6.19)	73.21
GEPA (Agrawal et al., 2025)	75.99(0.74)	89.88	57.14(3.86)	78.57
MIPRO (Opsahl-Ong et al., 2024)	74.40(1.46)	88.10	55.36(1.46)	80.36
CLEAR (ours)	**81.15**(2.48)	**91.67**	**66.67**(4.49)	**82.14**

Table 1: **AppWorld** experiments results. Task Goal Completion (TGC) and Scenario Goal Completion (SGC) on the TEST-N split are reported. Results are averaged over three runs (standard deviation in parentheses) except the Pass@3 metric.

HuggingFace. During execution for a new task $q_{\text{new}} \sim \mathcal{D}_{\text{test}}$, we find the most similar task q_j from the training set and retrieve the index according to $j = \arg\max_{1 \le i \le |\mathcal{D}_{\text{SFT}}|} \text{Sim}(E(q_i), E(q_{\text{new}}))$, where E denotes sentence embedding and $\text{Sim}(\cdot, \cdot)$ denotes the cosine similarity. The corresponding instruction c_j is then appended to the new task q_{new}. The execution agent subsequently operates on the augmented context $q_{\text{new}} \oplus c_j$. We refer to this approach as the RAG.

The compared methods. We also report results for ACE (Zhang et al., 2025) on AppWorld dataset. ACE models the context as an evolving playbook that accumulates and refines task-solving strategies through generation, reflection, and curation. To ensure a fair comparison, we adapt the official ACE GitHub repository[7] to the Strands Agents framework and use the same LLM, `Claude-Sonnet-4`, as in **CLEAR**. For GEPA and MIPROv2, we use the official implementations by `dspy` [8].

CLEAR. Given a new task q_{new}, we generate auxiliary context c using our CAM served via vLLM (Kwon et al., 2023): $c \sim \pi_\theta^C(q_{\text{new}})$. The execution agent π_θ^E then operates on the augmented description $q_{\text{new}} \oplus c$ to generate a trajectory and receive a reward.

5.3 Results on AppWorld

We report experiment results on AppWorld in this subsection. We use the TRAIN split as the training set $\mathcal{D}_{\text{train}}$ and the TEST-N split as the evaluation set $\mathcal{D}_{\text{test}}$. These two splits are disjoint and follow the official dataset partition defined in the original paper, ensuring that no data leakage occurs during evaluation. For the execution agent, we use the original system prompt, which is available in the official AppWorld repository[9].

Metrics. We use Task Goal Completion (TGC) and Scenario Goal Completion (SGC) rates from AppWorld (Trivedi et al., 2024) as our evaluation metrics. TGC is defined as the percentage of tasks for which the agent passes all evaluation tests provided by the AppWorld benchmark. SGC measures the percentage of task scenarios for which the agent passes all evaluation tests across every task within the scenario. We report TGC, SGC (averaged over 3 independent runs), and their pass@3 rates in Ta-

[7] https://github.com/ace-agent/ace

[8] https://github.com/stanfordnlp/dspy

[9] https://github.com/StonyBrookNLP/appworld/blob/main/experiments/prompts/react_code_agent/_legacy_instructions.txt

Method	Avg. Reward	Std.
Baseline	0.6799	0.0119
RAG	0.7252	0.0076
CLEAR (ours)	**0.7406**	0.0044

Table 2: **WebShop-40k** experiments results. Averaged reward on the test dataset is reported. Results are averaged over three runs with standard deviation also reported.

ble 1.

5.4 Results on WebShop-40k

As introduced earlier, we leverage Amazon Bedrock AgentCore for scalable rollout collection. This setup, however, imposes a constraint on Docker image size that is incompatible with the original WebShop benchmark: the full WebShop environment includes a local search index spanning millions of product items, which exceeds the image size limit enforced by AgentCore Runtime. To address this, we randomly sampled 40,000 items from the original product pool to construct a lightweight search index, forming the WebShop-40k variant. We then filtered the task set to retain only those tasks whose ground-truth target items exist within this 40k subset, ensuring reward calculation remains identical to the original benchmark formulation. We note that WebShop-40k may be inherently easier than the original benchmark, as the reduced search space lowers the difficulty of product retrieval. For the baseline agent, we curated a system prompt that describes the task objective, the available tools, and their corresponding descriptions.

Since ACE does not provide benchmark-specific prompts for WebShop dataset, we do not report ACE results on WebShop-40k. Other experimental settings are the same as AppWorld evaluation. The results are reported in Table 2.

5.5 Discussion

As shown in Table 1, **CLEAR** consistently outperforms all baselines across all models and all metrics, without using any benchmark-specific prompts in data generation and training pipeline. Especially compared to ACE, **CLEAR** achieves notable gains of $+6.75$ and $+7.74$ in TGC and SGC respectively, despite ACE using AppWorld-specific prompt for their reflector and curator. Similar improvements over the baselines are also observed on WebShop-40k dataset as shown in Table 2.

Ablation Study. To demonstrate all components in **CLEAR** are necessary, we perform ablation study in Section A. Table 3 in Section A shows that contrastive learning, SFT and RL each brings non-trivial performance improvement. See Section A for more details.

Latency Study. We present a latency study of CAM in Section B. As shown in Table 4, the additional overhead introduced by CAM is modest compared to the performance gains.

CAM Transferability. To study CAM transferability, we conducted additional study using `DeepSeek-V3.1` as π^E in Table 5 of Section D, while the entire CAM training data is generated from Claude model. We observe that CAM can still consistently outperform all baselines, despite the training and inference mismatch. See Section D for details.

6 Conclusion

LLM agents are increasingly used in sequential decision-making to complete complex tasks. In this paper, we propose **CLEAR**, a novel framework that trains a context augmentation model (CAM) to improve agent performance by generating task-relevant context and appending it to the prompt of the execution LLM agent. **CLEAR** first employs a reflection agent to perform contrastive analysis over past execution trajectories and summarize useful context for each observed task. These summaries are then used to train the CAM. Although **CLEAR** requires training a smaller CAM, it does not modify the parameters of the execution LLM agent. As a result, **CLEAR** can be applied to a wide range of LLM agent systems regardless of whether the underlying models are open-source or proprietary. Extensive experiments show that **CLEAR** consistently outperforms several strong baselines across multiple benchmarks.

Limitations

While **CLEAR** demonstrates strong empirical performance, several limitations remain.

First, the effectiveness of **CLEAR** relies on the quality and diversity of the collected experience replay buffer. If the replays are noisy, sparse, or lack sufficient successful examples, the reflection agent may fail to extract high-quality instructions, which in turn limits the performance of the learned context augmentation model.

Second, although **CLEAR** avoids modifying the execution agent, it introduces a non-trivial training pipeline involving reflection, SFT and RL, which can be computationally expensive and time-consuming. However, this cost is incurred only once during training. At inference time, **CLEAR** introduces minimal additional latency, making it practical for deployment in real-world applications. See section B for a study on inference latency.

References

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, and 1 others. 2025. Gepa: Reflective prompt evolution can outperform reinforcement learning. *arXiv preprint arXiv:2507.19457*.

Arash Ahmadian, Chris Cremer, Matthias Gallé, Marzieh Fadaee, Julia Kreutzer, Olivier Pietquin, Ahmet Üstün, and Sara Hooker. 2024. Back to basics: Revisiting reinforce-style optimization for learning from human feedback in llms. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12248–12267.

Marc-Antoine Allard, Arnaud Teinturier, Victor Xing, and Gautier Viaud. 2026. Experiential reflective learning for self-improving llm agents. *arXiv preprint arXiv:2603.24639*. ICLR 2026 MemAgents Workshop.

Anthropic. 2025. Introducing the model context protocol. https://www.anthropic.com/news/model-context-protocol.

Anthropic. 2025a. System card: Claude opus 4 & claude sonnet 4. Accessed: 2026-02-02.

Anthropic. 2025b. System card: Claude sonnet 4.5. Accessed: 2026-02-02.

Parth Asawa, Alan Zhu, Abby O'Neill, Matei Zaharia, Alexandros G Dimakis, and Joseph E Gonzalez. 2026. How to train your advisor: Steering black-box llms with advisor models. *arXiv preprint arXiv:2510.02453*.

Amanda Askell, Yuntao Bai, Anna Chen, Dawn Drain, Deep Ganguli, Tom Henighan, Andy Jones, Nicholas Joseph, Ben Mann, Nova DasSarma, and 1 others. 2021. A general language assistant as a laboratory for alignment. *arXiv preprint arXiv:2112.00861*.

Yuntao Bai, Andy Jones, Kamal Ndousse, Amanda Askell, Anna Chen, Nova DasSarma, Dawn Drain, Stanislav Fort, Deep Ganguli, Tom Henighan, and 1 others. 2022. Training a helpful and harmless assistant with reinforcement learning from human feedback. *arXiv preprint arXiv:2204.05862*.

Kevin Black, Noah Brown, Danny Driess, Adnan Esmail, Michael Equi, Chelsea Finn, Niccolo Fusai, Lachy Groom, Karol Hausman, Brian Ichter, and 1 others. 2024. π_0: A vision-language-action flow model for general robot control. *arXiv preprint arXiv:2410.24164*.

Hao Ding, Ziwei Fan, Ingo Guehring, Gaurav Gupta, Wooseok Ha, Jun Huan, Linbo Liu, Behrooz Omidvar-Tehrani, Shiqi Wang, and Hao Zhou. 2024. Reasoning and planning with large language models in code development. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 6480–6490.

Saman Forouzandeh, Wei Peng, Parham Moradi, Xinghuo Yu, and Mahdi Jalili. 2025. Learning hierarchical procedural memory for llm agents through bayesian selection and contrastive refinement. Accepted at AAMAS 2026.

Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yixin Dai, Jiawei Sun, Haofen Wang, Haofen Wang, and 1 others. 2023. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*, 2(1):32.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, and 1 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Michael Gutmann and Aapo Hyvärinen. 2010. Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 297–304, Chia Laguna Resort, Sardinia, Italy. PMLR.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, and 1 others. 2024. Metagpt: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations*.

Yaojie Hu, Qiang Zhou, Qihong Chen, Xiaopeng Li, Linbo Liu, Dejiao Zhang, Amit Kachroo, Talha Oz, and Omer Tripp. 2025. Qualityflow: An agentic workflow for program synthesis controlled by llm quality checks. *arXiv preprint arXiv:2501.17167*.

Yiqian Huang, Shiqi Zhang, and Xiaokui Xiao. 2025. Ket-rag: A cost-efficient multi-granular indexing framework for graph-rag. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2*, pages 1003–1012.

Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Sercan Arik, Dong Wang, Hamed Zamani, and Jiawei Han. 2025. Search-r1: Training llms to reason and leverage search engines with reinforcement learning. *arXiv preprint arXiv:2503.09516*.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

Jan Leike, David Krueger, Tom Everitt, Miljan Martic, Vishal Maini, and Shane Legg. 2018. Scalable agent alignment via reward modeling: a research direction. *arXiv preprint arXiv:1811.07871*.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, and 1 others. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33:9459–9474.

Cheng Li, Mingyang Zhang, Qiaozhu Mei, Weize Kong, and Michael Bendersky. 2024. Learning to rewrite prompts for personalized text generation. In *Proceedings of the ACM Web Conference 2024*, pages 3367–3378.

Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. 2023. Camel: Communicative agents for "mind" exploration of large language model society. In *Advances in Neural Information Processing Systems*, volume 36.

Vijay Lingam, Behrooz Omidvar Tehrani, Sujay Sanghavi, Gaurav Gupta, Sayan Ghosh, Linbo Liu, Jun Huan, and Anoop Deoras. 2025. Enhancing language model agents using diversity of thoughts. In *The Thirteenth International Conference on Learning Representations*.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, and 1 others. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.

Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, and 1 others. 2025a. Deepseek-v3. 2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*.

Linbo Liu, Xinle Liu, Qiang Zhou, Lin Chen, Yihan Liu, Hoan Nguyen, Behrooz Omidvar-Tehrani, Xi Shen, Jun Huan, Omer Tripp, and 1 others. 2025b. Migrationbench: Repository-level code migration benchmark from java 8. *arXiv preprint arXiv:2505.09569*.

Xinbei Ma, Yeyun Gong, Pengcheng He, Hai Zhao, and Nan Duan. 2023. Query rewriting in retrieval-augmented large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 5303–5315.

Zhuang Ma and Michael Collins. 2018. Noise contrastive estimation and negative sampling for conditional models: Consistency and statistical efficiency. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3698–3707, Brussels, Belgium. Association for Computational Linguistics.

Lingrui Mei, Jiayu Yao, Yuyao Ge, Yiwei Wang, Baolong Bi, Yujun Cai, Jiazhi Liu, Mingyu Li, Zhong-Zhi Li, Duzhen Zhang, and 1 others. 2025. A survey of context engineering for large language models. *arXiv preprint arXiv:2507.13334*.

Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, and 1 others. 2021. Webgpt: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332*.

Team Olmo, Allyson Ettinger, Amanda Bertsch, Bailey Kuehl, David Graham, David Heineman, Dirk Groeneveld, Faeze Brahman, Finbarr Timbers, Hamish Ivison, and 1 others. 2025. Olmo 3. *arXiv preprint arXiv:2512.13961*.

OpenAI. 2025. Introducing operator. https://openai.com/index/introducing-operator/.

Krista Opsahl-Ong, Michael J Ryan, Josh Purtell, David Broman, Christopher Potts, Matei Zaharia, and Omar Khattab. 2024. Optimizing instructions and demonstrations for multi-stage language model programs. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 9340–9366.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, and 1 others. 2022. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744.

Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. 2024. Gorilla: Large language model connected with massive apis. *Advances in Neural Information Processing Systems*, 37:126544–126565.

Wenjun Peng, Guiyang Li, Yue Jiang, Zilong Wang, Dan Ou, Xiaoyi Zeng, Derong Xu, Tong Xu, and Enhong Chen. 2024. Large language model based long-tail query rewriting in taobao search. In *Companion Proceedings of the ACM Web Conference 2024*, pages 20–28.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng

Su, Xin Cong, and 1 others. 2024. Chatdev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*.

Cheng Qian, Emre Can Acikgoz, Qi He, Hongru Wang, Xiusi Chen, Dilek Hakkani-Tür, Gokhan Tur, and Heng Ji. 2025. Toolrl: Reward is all tool learning needs. *arXiv preprint arXiv:2504.13958*.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. 2023. Direct preference optimization: Your language model is secretly a reward model. *Advances in neural information processing systems*, 36:53728–53741.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *Advances in neural information processing systems*, 36:68539–68551.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Zhihong Shao, Yeyun Gong, Yelong Shen, Minlie Huang, Nan Duan, and Weizhu Chen. 2023. Enhancing retrieval-augmented large language models with iterative retrieval-generation synergy. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 9248–9274.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, and 1 others. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. 2024. Hybridflow: A flexible and efficient rlhf framework. *arXiv preprint arXiv: 2409.19256*.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Sheshadri, Karthik Narasimhan, Shunyu Yao, and 1 others. 2023. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36.

Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, and 1 others. 2025. Openai gpt-5 system card. *arXiv preprint arXiv:2601.03267*.

Mirac Suzgun, Mert Yuksekgonul, Federico Bianchi, Dan Jurafsky, and James Zou. 2025. Dynamic cheatsheet: Test-time learning with adaptive memory. *arXiv preprint arXiv:2504.07952*.

Yashar Talebirad and Amirhossein Nadiri. 2023. Multi-agent collaboration: Harnessing the power of intelligent llm agents. *arXiv preprint arXiv:2306.03314*.

Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, and 1 others. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*.

Kimi Team, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, and 1 others. 2025. Kimi k2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. 2024. Appworld: A controllable world of apps and people for benchmarking interactive coding agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 16022–16076.

Aaron van den Oord, Yazhe Li, and Oriol Vinyals. 2019. Representation learning with contrastive predictive coding. *Preprint*, arXiv:1807.03748.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, and 1 others. 2024a. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):186345.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024b. Executable code actions elicit better llm agents. In *Forty-first International Conference on Machine Learning*.

Xingyao Wang, Simon Rosenberg, Juan Michelini, Calvin Smith, Hoang Tran, Engel Nyst, Rohit Malhotra, Xuhui Zhou, Valerie Chen, Robert Brennan, and Graham Neubig. 2025. The openhands software agent sdk: A composable and extensible foundation for production agents. *Preprint*, arXiv:2511.03690.

Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, and 1 others. 2022a. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022b. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, and 1 others. 2024. Autogen: Enabling next-gen llm applications via multi-agent conversation. In *Conference on Language Modeling*.

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, and 1 others. 2025. The rise and potential of large language model based agents: A survey. *Science China Information Sciences*, 68(2):121101.

Chunqiu Steven Xia, Zhe Wang, Yan Yang, Yuxiang Wei, and Lingming Zhang. 2025. Live-swe-agent: Can software engineering agents self-evolve on the fly? *arXiv preprint arXiv:2511.13646*.

Shitao Xiao, Zheng Liu, Peitian Zhang, and Niklas Muennighoff. 2023. C-pack: Packaged resources to advance general chinese embedding. *Preprint*, arXiv:2309.07597.

Zhichao Xu, Zhiqi Huang, Shengyao Zhuang, and Vivek Srikumar. 2025a. Distillation versus contrastive learning: How to train your rerankers. In *Proceedings of the 14th International Joint Conference on Natural Language Processing and the 4th Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics*, pages 564–578, Mumbai, India. The Asian Federation of Natural Language Processing and The Association for Computational Linguistics.

Zhichao Xu, Fengran Mo, Zhiqi Huang, Crystina Zhang, Puxuan Yu, Bei Wang Phillips, Jimmy Lin, and Vivek Srikumar. 2026. A survey of model architectures in information retrieval. *Transactions on Machine Learning Research*. Survey Certification.

Zhichao Xu, Shengyao Zhuang, Xueguang Ma, Bingsen Chen, Yijun Tian, Fengran Mo, Jie Cao, and Vivek Srikumar. 2025b. Rethinking on-policy optimization for query augmentation. *Preprint*, arXiv:2510.17139.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. 2022. Webshop: Towards scalable real-world web interaction with grounded language agents. *Advances in Neural Information Processing Systems*, 35:20744–20757.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023a. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023b. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*.

Hongzhuo Yu, Fei Zhu, Guo-Sen Xie, and Ling Shao. 2026. Self-consolidation for self-evolving agents.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, and 1 others. 2025. Dapo: An open-source llm reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*.

Wenhao Yu, Dan Iter, Shuohang Wang, Yichong Xu, Mingxuan Ju, Soumya Sanyal, Chenguang Zhu, Michael Zeng, and Meng Jiang. 2022. Generate rather than retrieve: Large language models are strong context generators. *arXiv preprint arXiv:2209.10063*.

Yu Yue, Yufeng Yuan, Qiying Yu, Xiaochen Zuo, Ruofei Zhu, Wenyuan Xu, Jiaze Chen, Chengyi Wang, TianTian Fan, Zhengyin Du, and 1 others. 2025. Vapo: Efficient and reliable reinforcement learning for advanced reasoning tasks. *arXiv preprint arXiv:2504.05118*.

Junzi Zhang, Jongho Kim, Brendan O'Donoghue, and Stephen Boyd. 2021. Sample efficient reinforcement learning with reinforce. In *Proceedings of the AAAI conference on artificial intelligence*, volume 35, pages 10887–10895.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. 2025. Agentic context engineering: Evolving contexts for self-improving language models. *Preprint*, arXiv:2510.04618.

Wenzheng Zhang and Karl Stratos. 2021. Understanding hard negatives in noise contrastive estimation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1090–1101, Online. Association for Computational Linguistics.

Chujie Zheng, Shixuan Liu, Mingze Li, Xiong-Hui Chen, Bowen Yu, Chang Gao, Kai Dang, Yuqiong Liu, Rui Men, An Yang, and 1 others. 2025. Group sequence policy optimization. *arXiv preprint arXiv:2507.18071*.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, Zhangchi Feng, and Yongqiang Ma. 2024. Llamafactory: Unified efficient fine-tuning

of 100+ language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, Bangkok, Thailand. Association for Computational Linguistics.

A Ablation Study

We perform some ablation study on **CLEAR** framework in this section. We will show that all three phases in **CLEAR** are necessary and removing any of them might result in suboptimal performance of CAM. All the following experiments are conducted on AppWolrd using `Claude-Sonnet-4` for π^E.

First, we show that the RL phase in **CLEAR** is necessary. We remove RL phase and use π^C_{SFT} without RL as CAM. We report the performance of π^C_{SFT} in experiment 3 in Table 3. Compared with a full **CLEAR** framework with SFT + RL (experiment 4), π^C_{SFT} significantly degrade all metrics in TGC and SGC.

Next, we show that contrastive learning (CL) is necessary. To illustrate this, we curate an SFT training dataset $\mathcal{D}_{\text{SFT_no_CL}}$ using only one trajectory for each task. We then perform SFT using $\mathcal{D}_{\text{SFT_no_CL}}$ for CAM and report the performance in experiment 2 in Table 3. Compared with SFT using CL (experiment 3), SFT without CL significantly underperforms experiment 3, which shows that CL plays an important role in increasing SFT data quality.

Finally, we show that SFT is necessary. We use a `Qwen/Qwen3-32B` model directly downloaded from HuggingFace as CAM without any fine-tuning and report the performance in experiment 1 in Table 3. Comparing experiment 1 and 3, we see that using data from CL to SFT a CAM has significant performance gain over using an untuned `Qwen/Qwen3-32B` model as CAM.

B Latency Study

In this section, we present latency study for triggering a CAM. We report the averaged task execution time, averaged number of turns of the execution agent π^E, averaged throughput of CAM, and averaged latency for invoking CAM. The average is taken across AppWorld TEST-N split. The CAM is hosted via vllm on 8 NVIDIA B200 GPUs.

From Table 4, we see that on average, **CLEAR** increases number of turns by 1.2 and triggers a latency of 1.2 second to invoke CAM. These two components sum up to a 13.8 second increase in terms of task run time. Overall, this additional overhead is modest compared to the performance gains achieved by **CLEAR**.

C Comparison to RAG

We discuss the similarities and differences between **CLEAR** and RAG. Our augmentation model $\pi^C(\cdot)$ can be effectively viewed as a *generative retrieval* model, following the idea of generate-then-read (Yu et al., 2022). For each task, it *generates* the most useful context from its internal parameters, rather than *retrieving* the most similar context from an external knowledge base.

The key difference lies in how the retrieved information is used. In RAG, knowledge items are retrieved as-is, and the execution agent must reason over them to determine how they should be applied to the current task, assuming the current task is not available in knowledge base. This is true because the knowledge base is established using the training set, which is disjoint from the test set. In contrast, **CLEAR** shifts this reasoning burden to the augmentation model π^C, which generates context that is already tailored to the new query. As a result, the generated context c is directly actionable for the execution agent π^E, reducing the amount of additional reasoning required from π^E, particularly when the execution model is relatively weak.

This statement is supported by the experiments with `DeepSeek-V3.1` on AppWorld, as shown in Table 5. Compared with `Claude-Sonnet-4`, `DeepSeek-V3.1` is generally considered less capable (see their respective model cards). Under this setting, the RAG baseline with `DeepSeek-V3.1` even underperforms the vanilla baseline, suggesting that simply retrieving items by embedding similarity is noisy when the underlying model lacks strong reasoning ability. In contrast, **CLEAR** improves performance by generating task-specific context that is already adapted to the new query. Similar performance degradation can be observed for ACE, whose playbook is curated from training trajectories. As a result, the execution agent π^E must still perform additional reasoning to determine how the retrieved instructions apply to the current task.

D CAM Transferability

The objective of CAM is to provide auxiliary context and can be detached from the task execution agent π^E. In this section, we study whether a trained CAM can be applied to a different π^E with-

Experiment ID	Method	TGC ↑		SGC ↑	
		Avg	Pass@3	Avg	Pass@3
1	`Qwen/Qwen3-32B`	67.40(2.95)	83.33	48.81(4.49)	66.07
2	SFT without CL	68.65 (2.68)	88.69	42.86 (4.72)	67.86
3	SFT with CL	74.21(1.72)	90.48	50.60(2.73)	71.43
4	**CLEAR**	**81.15**(2.48)	**91.67**	**66.67**(4.49)	**82.14**

Table 3: Ablation study on **AppWorld** TEST-N split. For the CAM, we use the following variant: (1) a `Qwen/Qwen3-32B` model without SFT and RL as in experiment 1. (2) a Qwen3-32B model after SFT on $\mathcal{D}_{\text{SFT_no_CL}}$ (no RL) as in experiment 2. (3) a Qwen3-32B model after SFT with CL (no RL) as in experiment 3. (4) full **CLEAR** framework as in experiment 4. Results are averaged over three runs (standard deviation in parentheses) except the Pass@3 metric.

Method	Run time (s)	# of turns	CAM throughput (tokens/s)	CAM latency (s)
Baseline	63.4	17.3	NA	NA
CLEAR (ours)	77.2	18.5	69.5	1.21

Table 4: CAM Latency study. Task run time, number of turns of the execution agent π^E, throughput of CAM, and latency for invoking CAM. All numbers are averaged over AppWorld TEST-N split. The model for π^E is `Claude-Sonnet-4`.

out retraining.

Recall that the training dataset $\mathcal{D}_{\text{train}}$ is generated by contrastive analysis of the replay buffer Γ, which is generated by the execution agent π^E powered by `Claude-Sonnet-4`. The reflection agent π^R that analyzes Γ is also powered by `Claude-Sonnet-4`. Furthermore, during Phase 2 RL training, the execution agent π^E used for reward computation also uses `Claude-Sonnet-4` as its foundation model. Consequently, the entire CAM training pipeline relies solely on trajectories and feedback generated by the Claude model.

Despite this, the trained CAM demonstrates strong transferability. At test time, it still provides significant performance gains when the execution agent π^E is replaced by a different model, such as `DeepSeek-V3.1`. For example, averaged TGC and SGC increase by 1.68 and 5.35 over the baseline respectively, as shown in Table 5. This result suggests that once trained with the **CLEAR** framework, the CAM can generalize across different execution agents without requiring retraining.

E Experiment Setting

We introduce our experiment setting for all three phases: agentic reflection, SFT, and RL.

E.1 Agentic Data Collection

Reflection Agent. We use the Strands Agents framework together with `Claude-Sonnet-4` to build a reflection agent for contrastive analysis. The full prompt used for the reflection agent is provided in Section F.

If all $m = 6$ runs of a task are processed by a single reflection pass, the resulting dataset $\mathcal{D}_{\text{SFT}}$ would have the same size as the training dataset $\mathcal{D}_{\text{train}}$, which is too small to effectively fine-tune LLMs with billions of parameters.

To increase the amount of data, for each task we sample subsets of 3 runs from the 6 collected trajectories. The reflection agent only analyzes the 3 selected runs. This process can be repeated for $\binom{6}{3} = 20$ times, which effectively expands the training dataset by a factor of 20.

E.2 Supervised Fine-Tuning

We further randomly split $\mathcal{D}_{\text{SFT}}$ into 80% for training and 20% for validation. The CAM $\pi^C_\theta(\cdot)$ is initialized from a `Qwen/Qwen3-32B` model (Yang et al., 2025), downloaded from HuggingFace[10]. We perform full-parameter fine-tuning for 5 epochs using 8 NVIDIA B200 GPUs. Training is conducted in bf16 precision with a learning rate of 1×10^{-5} and a warm-up ratio of 0.05. The supervised fine-tuning is implemented using the LLAMAFACTORY

[10] https://huggingface.co/Qwen/Qwen3-32B

Model for π^E	Method	TGC ↑		SGC ↑	
		Avg	Pass@3	Avg	Pass@3
DeepSeek-V3.1	Baseline	41.27(3.05)	64.29	19.05(2.06)	26.79
	RAG	33.33(1.57)	53.57	10.71(4.72)	21.43
	ACE	32.54(1.50)	53.57	13.69(1.03)	28.57
	CLEAR (ours)	**42.95**(2.85)	**66.07**	**24.40**(4.49)	**33.93**

Table 5: **AppWorld** experiments results. Task Goal Completion (TGC) and Scenario Goal Completion (SGC) on the TEST-N split are reported. Results are averaged over three runs (standard deviation in parentheses) except the Pass@3 metric.

framework (Zheng et al., 2024).

E.3 Reinforcement Learning with GRPO

We perform reinforcement learning on the augmentation model π^C_{SFT} using the GRPO algorithm, implemented with the Verl framework (Sheng et al., 2024). Training is conducted for 15 epochs on the train dataset using 8 NVIDIA B200 GPUs.

As described in Section 4.2, computing the GRPO reward requires multi-turn interactions between the task execution agent π^E_θ and the benchmark environment M defined in Equation (1). To efficiently compute rewards in batch, we also leverage Amazon Bedrock AgentCore Runtime, which bootstraps multiple containers in parallel to support high-concurrency rollout execution hence reward computation. Additional hyperparameter configurations for GRPO are provided below.

```
python3 -m verl.trainer.main_ppo \
    algorithm.adv_estimator=grpo \
    data.train_files=<path/to/train.parquet> \
    data.val_files=<path/to/train.parquet> \
    data.train_batch_size=4 \
    data.max_prompt_length=4096 \
    data.max_response_length=1024 \
    data.filter_overlong_prompts=True \
    data.truncation='error' \
    actor_rollout_ref.model.path=<model_path> \
    actor_rollout_ref.actor.optim.lr=1e-6 \
    actor_rollout_ref.model.use_remove_padding=
        True \
    actor_rollout_ref.actor.ppo_mini_batch_size=2
         \
    actor_rollout_ref.actor.
        ppo_micro_batch_size_per_gpu=1 \
    actor_rollout_ref.actor.use_kl_loss=True \
    actor_rollout_ref.actor.kl_loss_coef=0.001 \
    actor_rollout_ref.actor.kl_loss_type=
        low_var_kl \
    actor_rollout_ref.actor.entropy_coeff=0 \
    actor_rollout_ref.model.
        enable_gradient_checkpointing=True \
    actor_rollout_ref.actor.fsdp_config.
        param_offload=False \
    actor_rollout_ref.actor.fsdp_config.
        optimizer_offload=False \
    actor_rollout_ref.rollout.
        log_prob_micro_batch_size_per_gpu=2 \
    actor_rollout_ref.rollout.
        tensor_model_parallel_size=2 \
    actor_rollout_ref.rollout.name=vllm \
    actor_rollout_ref.rollout.
        gpu_memory_utilization=0.5 \
    actor_rollout_ref.rollout.n=4 \
    actor_rollout_ref.ref.
        log_prob_micro_batch_size_per_gpu=2 \
    actor_rollout_ref.ref.fsdp_config.
        param_offload=True \
    algorithm.use_kl_in_reward=False \
    trainer.critic_warmup=0 \
    trainer.n_gpus_per_node=8 \
    trainer.nnodes=1 \
    trainer.save_freq=20 \
    trainer.test_freq=20 \
    trainer.total_epochs=15 \
    actor_rollout_ref.rollout.checkpoint_engine.
        update_weights_bucket_megabytes=4096 \
    custom_reward_function.path=<
        reward_function_path> $@
```

F Prompt for Reflection Agent

We provide the system prompt and user/task prompt for the reflection agent $\pi^R(\cdot)$. They are universal across different benchmarks.

F.1 System Prompt

System Prompt for the Reflection Agent

````
Your task is to analyze LLM agent execution
    trajectories and generate guidance using
    contrastive learning.

## CRITICAL: Context Window Management -
    READ THIS FIRST

**WARNING**: Trajectory files can be
    EXTREMELY large (100K-500K+ tokens).
    Reading entire files will cause
    out-of-context-window errors and crash
    the session.

**MANDATORY FIRST STEP**: Before reading ANY
    file content, you MUST check its size:
```bash
# ALWAYS run this first for any file
````
````

```
ls -lh <file_path>
wc -l <file_path>  # Count lines
wc -c <file_path>  # Count bytes
```

Size Guidelines:
- Files < 10KB: Safe to read with `cat` or `jq`
- Files 10KB-100KB: Use `head`, `tail`, or targeted `jq` queries
- Files > 100KB: ONLY use regex/grep patterns, NEVER read full content
- Files > 1MB: Use aggressive filtering with grep + head limits

NEVER DO THIS:
- `cat large_file.json` - Will overflow context
- `jq '.' large_file.json` - Will overflow context
- Reading entire message arrays without limits

ALWAYS DO THIS:
- Check file size FIRST
- Use regex patterns with grep to search
- Limit output with `head -n` or `tail -n`
- Use `jq` with specific field selectors, not full dumps

Available Tools

You have access to the `shell` tool to execute bash commands.

Step 1: ALWAYS Check File Sizes First
```bash
# MANDATORY: Run these commands BEFORE any content extraction
ls -lh <trajectory_folder>/*.json
du -sh <trajectory_folder>

# List files sorted by size
ls -lhS <trajectory_folder>/*.json | head -20
```

Step 2: Use Regex Patterns for Searching (Primary Method)
```bash
# Search for error patterns using regex - MOST EFFICIENT
grep -E "error|Error|ERROR|exception|Exception|failed|Failed" <file_path> | head -30
grep -oE '"error":\\s*"[^"]*"' <file_path> | head -20
grep -oE '"success":\\s*(true|false)' <file_path> | head -10

# Search for specific patterns with context
grep -E -B2 -A2 "error|exception" <file_path> | head -50

# Count occurrences of patterns
grep -cE "error|failed|exception" <file_path>

# Find tool calls and their patterns
grep -oE '"tool_name":\\s*"[^"]*"' <file_path> | sort | uniq -c | sort -rn

# Extract specific JSON values with regex
grep -oE '"reward":\\s*[0-9.-]+' <file_path>
grep -oE '"task_id":\\s*"[^"]*"' <file_path>

# Search for repeated patterns (loops, stuck behavior)
grep -oE '"content":\\s*"[^"]{0,100}' <file_path> | sort | uniq -c | sort -rn | head -20
```

Step 3: Targeted jq Queries (Only for Small Extractions)
```bash
# ONLY use jq for small, targeted extractions
jq 'keys' <file_path>  # Structure overview (safe)
jq '.data.messages | length' <file_path>  # Count messages (safe)
jq '.data.messages[0]' <file_path>  # First message only
jq '.data.messages[-1]' <file_path>  # Last message only
jq '{reward: .data.reward, success: .data.metrics.success}' <file_path>  # Summary stats

# DANGEROUS - Only use with head limit:
jq -r '.data.messages[].content // empty' <file_path> 2>/dev/null | head -50
```

Contrastive Learning Strategy

1. **Identify successful vs failed traces** using reward/success metrics
2. **Compare agent behaviors** between successful and failed cases
3. **Find patterns** that differentiate success from failure
4. **Generate insights** about what works and what doesn't

Output Format

Your final output should be a task-specific guidance in the following format:
<guidance>
The strategy for completing this task is ...
</guidance>

F.2 User Prompt

User Prompt for the Reflection Agent

```
You are given a folder that contains multiple LLM agent trajectory traces for solving the same task.
They are a mix of successful and failed executions in general, indicated by reward/success metrics in the traces, but it's also possible that all traces are successful or failed.
Please analyze multiple LLM agent trajectories and generate a guidance using contrastive learning.
```

```
## Trajectory Folder
Path: {{ traces_folder }}

This folder contains multiple trajectory
    JSON files - a mix of successful and
    failed agent executions. You can work
    outside this folder.

## CRITICAL: Instructions (Follow in Order)

**Step 1: CHECK FILE SIZES FIRST
    (MANDATORY)**
Before reading ANY trajectory content, run:
```bash
ls -lh {{ traces_folder }}/*.json
    2>/dev/null || ls -lh {{ traces_folder }}
find {{ traces_folder }} -name "*.json"
    -exec ls -lh {} \; | head -20
```
This prevents context window overflow errors.

**Step 2: IDENTIFY SUCCESS VS FAILURE
    (CONTRASTIVE SETUP)**
Extract reward/success metrics from all
    files to categorize them:
```bash
# Get reward values from all trajectory files
for f in {{ traces_folder }}/*.json; do
    echo -n "$f: "
    grep -oE '"reward":\s*[0-9.-]+' "$f" |
    head -1
done 2>/dev/null | head -30

# Or use jq if files are small
find {{ traces_folder }} -name "*.json"
    -size -100k -exec sh -c 'echo -n "{}: "
    && jq -r ".data.reward // .reward //
    \"N/A\"" "{}"' \; 2>/dev/null | head -30
```

Categorize traces into:
- **Successful traces**: High reward (e.g.,
    reward > 0.5 or success=true)
- **Failed traces**: Low reward (e.g.,
    reward <= 0 or success=false)

Step 3: ANALYZE SUCCESSFUL TRACES
For successful traces, identify:
- What actions/strategies led to success
- How the agent handled challenges
- Decision-making patterns
- Skip this step if there is no successful
    trace

```bash
# Example: Analyze a successful trace
    (adjust filename)
grep -oE '"tool_name":\s*"[^"]*"'
    <successful_trace.json> | sort | uniq -c
    | sort -rn
grep -E -B2 -A2 "success|complete|done"
    <successful_trace.json> | head -50
```

Step 4: ANALYZE FAILED TRACES
For failed traces, identify:
- What went wrong (errors, loops, wrong
    decisions)
- Where the agent got stuck
- Common failure patterns
- Skip this step if there is no failed traces

```bash
# Example: Analyze a failed trace (adjust
    filename)
grep -E
    "error|Error|failed|Failed|exception"
    <failed_trace.json> | head -30
grep -oE '"content":\s*"[^"]{0,100}'
    <failed_trace.json> | sort | uniq -c |
    sort -rn | head -20
```

Step 5: CONTRASTIVE COMPARISON
Compare successful vs failed traces to
    identify:
1. **Differentiating behaviors**: What do
    successful agents do that failed ones
    don't?
2. **Common pitfalls**: What mistakes do
    failed agents make?
3. **Recovery strategies**: How do
    successful agents recover from errors?
4. **Decision quality**: How do successful
    agents make better choices?

Step 6: GENERATE TASK-SPECIFIC GUIDANCE
Based on your contrastive analysis, generate
    a step-by-step guidance for this task.
    The goal is to add this guidance at the
    beginning of the task, so that the agent
    can always successfully complete the
    task. Be concise.

Output Requirements

After your analysis, output the
    task-specific guidance in the following
    format:

<guidance>
The strategy for completing this task is ...
</guidance>
```

G Prompt for AppWorld

For AppWorld dataset, We use the official system prompt released at https://github.com/StonyBrookNLP/appworld/blob/main/experiments/prompts/react_code_agent/_legacy_instructions.txt for π^E. For completeness, we include it below.

User Prompt for the Reflection Agent

```
USER:
I am your supervisor and you are a super
    intelligent AI Assistant whose job is to
    achieve my day-to-day tasks completely
    autonomously.

To do this, you will need to interact with
    app/s (e.g., spotify, venmo etc) using
    their associated APIs on my behalf. For
    this you will undertake a *multi-step
```

```
    conversation* using a python REPL
    environment. That is, you will write the
    python code and the environment will
    execute it and show you the result,
    based on which, you will write python
    code for the next step and so on, until
    you've achieved the goal. This
    environment will let you interact with
    app/s using their associated APIs on my
    behalf.

Here are three key APIs that you need to
    know to get more information

# To get a list of apps that are available
    to you.

```python
print(apis.api_docs.show_app_descriptions())
```

To get the list of apis under any app
    listed above, e.g. spotify

```python
print(apis.api_docs.show_api_descriptions(
app_name='spotify'))
```

To get the specification of a particular
    api, e.g. spotify app's login api

```python
print(apis.api_docs.show_api_doc(
app_name='spotify', api_name='login'))
```

Each code execution will produce an output
    that you can use in subsequent calls.
    Using these APIs, you can now generate
    code, that I will execute, to solve the
    task. Let's start with the task

<An in-context learning example. Omitted for
    better readability.>

USER:
Key instructions:
(1) Make sure to end code blocks with ```
    followed by a newline(\n).

(2) Remember you can use the variables in
    your code in subsequent code blocks.

(3) Remember that the email addresses,
    access tokens and variables (e.g.
    spotify_password) in the example above
    are not valid anymore.

(4) You can use the "supervisor" app to get
    information about my accounts and use
    the "phone" app to get information about
    friends and family.

(5) Always look at API specifications (using
    apis.api_docs.show_api_doc) before
    calling an API.

(6) Write small chunks of code and only one
    chunk of code in every step. Make sure
    everything is working correctly before
    making any irreversible change.

(7) Many APIs return items in "pages". Make
    sure to run through all the pages by
    looping over `page_index`.

(8) Once you have completed the task, make
    sure to call
    apis.supervisor.complete_task(). If the
    task asked for some information, return
    it as the answer argument, i.e. call
    apis.supervisor.complete_task(answer=
<answer>). Many tasks do not require an
    answer, so in those cases, just call
    apis.supervisor.complete_task() i.e. do
    not pass any argument.

USER:
Using these APIs, now generate code to solve
    the actual task:

My name is: {{ main_user.first_name }} {{
    main_user.last_name }}. My personal
    email is {{ main_user.email }} and phone
    number is {{ main_user.phone_number }}.
Task: {{ input_str }}
```

H WebShop Prompt

We provide the systemp prompt of the execution agent $\pi^E(\cdot)$ for WebShop.

User Prompt for the Reflection Agent

```
You are an expert online shopping assistant
    designed to help users find and purchase
    products that match their specific
    requirements on an e-commerce website.

Your goal is to navigate the shopping
    website efficiently and complete
    purchase tasks by:
1. Understanding the user's product
    requirements completely
2. Searching for relevant products using
    appropriate keywords
3. Evaluating product details to ensure they
    match ALL requirements
4. Selecting the correct product options
    (size, color, etc.)
5. Completing the purchase by clicking "buy
    now"

## Available Tools

You have access to three tools to interact
    with the website:

### 1. search(query: str)
Search for products using space-separated
    keywords.
- Keywords are matched against product
    titles, descriptions, and attributes
```

```
- Use specific keywords that capture the
    essential requirements
- Examples: "red shirt large", "wooden toy",
    "laptop bag waterproof"

### 2. click(element: str)
Click on elements like product links,
    buttons, or options.
- Product ASINs (e.g., "b09mnnbdtj") - to
    view product details
- Navigation: "next >", "< prev", "back to
    search"
- Options: "red", "blue", "small", "medium",
    "large", etc.
- Tabs: "description", "features",
    "reviews", "attributes"
- Action: "buy now" - to complete purchase
    (USE THIS WHEN READY!)

### 3. get_available_actions()
Get a list of all clickable elements on the
    current page.
- Use this when you're unsure what actions
    are available
- Helpful when clicks are failing or you
    need to see all options
- Returns: has_search_bar boolean and list
    of clickable elements

## Task Completion Strategy

1. **Understand Requirements**: Carefully
    read the task to identify all product
    requirements (type, color, size,
    features, etc.)

2. **Search Effectively**:
    - Use relevant keywords that match the
    product requirements
    - Start with general terms, then refine
    if needed
    - Review search results to find
    promising products

3. **Evaluate Products**:
    - Click on product ASINs to view details
    - Check description, features, and
    attributes tabs
    - Verify the product matches ALL
    requirements from the task

4. **Select Options**:
    - If the product has options (size,
    color), select the ones matching
    requirements
    - Use get_available_actions() if you
    need to see what options are available

5. **Complete Purchase**:
    - Once you've found a product that
    matches all requirements and selected
    appropriate options
    - Click "buy now" to complete the task
    - This will end the episode and you'll
    receive a reward

## Important Tips
```

```
- Be thorough: Verify all requirements
    before purchasing
- If search results don't match, try
    different keywords
- Use get_available_actions() when uncertain
    about next steps
- Don't buy products that only partially
    match requirements
- Read product details carefully - don't
    assume based on title alone

## Example Workflow

Task: "Find a red shirt in size large"

1. search("red shirt large")
2. Click on promising product ASIN
3. Check description/features to verify it's
    actually red
4. Look for size options and click("large")
5. Verify all requirements are met
6. click("buy now")

Now let's complete your shopping task!
```